COMSC-210 Lab 12
Client-Server Computer Simulations

GOOD PROGRAMMING PRACTICES *show / hide*

ABOUT THIS ASSIGNMENT
One particularly useful application of data structures in programming is in the simulation of physical systems. Simulations have wide application, from the serious, such as simulations of nuclear power plant accidents and atomic bomb explosions, to the entertaining, such as real-time games with life-like Newtonian motion. Events are stored in priority queues, so that the next event is always readily accessible. Piles are represented in stacks, for entering and backing out of structures. Transaction processing is managed with queues, for first-come, first-served processing.

In this assignment you will write a simulation of a "client-server" customer service operation with multiple servers and a single **wait queue**.

After you complete this lab assignment, post the required files to the COMSC server so that you receive the proper credit for this 50-point lab. Your posted files are due at midnight of the evening of the due date indicated in the course outline. Use the "Submit your FA2015 work" link on the class website to post your file for this lab, using the **lab12** folder provided.

LAB 12: Write A Customer Service Simulation `[ Simulation.cpp ]`
Purpose. In this lab you will apply parts of the STL to the solution of a computer simulation problem.

Requirements. Write **`Simulation.cpp`** to perform a minute-by-minute simulation based on 6 inputs (to be read from a text file named simulation.txt, as numbers *only*, one per line, in this order):

1. the number of servers (1 or more, whole number)
2. the average arrival rate of customers, per minute (greater than 0.0, ***floating point***)
3. the maximum length of the **wait queue** (1 or more, whole number)
4. the minimum service time interval, in minutes (1 or more, whole number)
5. the maximum service time interval, in minutes (>=minimum service time interval, whole number)
6. the clock time at which new arrivals stop, in minutes (>0, whole number)

Echo the above 6 values in your output, well-labeled, as modeled in the sample output below.

Each minute's output should include the following:

- the clock time -- that is, the amount of time since the start of the simulation -- in minutes (whole number)
- a visual representation of the **wait queue**
- the average time interval from arrival to end of service per served customer, if the #of served customers is greater than zero (floating point number with 3 decimal places)
- after clock time zero: the rate at which customers are denied due to a full **wait queue** (floating point), computed as the count of denied customers divided by the clock time

The simulation should pause after each minute's summary is outputted, enabling the user to press ENTER to continue to the next minute, or to type `x` or `X` and then press ENTER to end the simulation. Do *not* let the simulation end after new arrivals stop arriving -- only end the program upon the user's `x` or `X`.

Here are the specs:

- Create a struct or class to represent a customer. Include these data members: arrival time, service start time, and service end time, all as whole numbers. "Arrival time" is the clock time that the customer arrives to be placed in the **wait queue**. "Service start time" is the clock time that the customer's service starts, either by arriving when a server is idle, or by moving out of the **wait queue**. Service time does not have to be initialized -- it's set only when service actually begins for the customer. "Service end time" is the clock time that the customer's service ends. End time does not have to be initialized -- it's set only when service actually ends for the customer.
- In the customer object, also include an ID tag for the customer, stored as a single letter of the alphabet, A-Z. Assign A to the first-created customer, B to the next, and so on. After the 26th customer is assigned Z, start the IDs over again -- that is, assign A to the 27th customer. HINT: use a variable to count the #of customers ever created (e.g., n, and get the corresponding ID code with (char) ('A' + (n++ % 26)).
- Customers arrive at the specified average arrival rate from the beginning of the simulation until the specified clock time at which new arrivals stop. After that time there are no new arrivals, but the simulation continues, allowing the **wait queue** to empty.
- Create a struct or class to represent a service event, that has an overloaded operator-less-than so that events can be stored in a priority queue based on their event clock time, lo-to-hi. Include 2 data members: clock time (whole number) for the event (i.e., completed service) to take place, and the server number (0-based index).
- Read 6 input values from a text file `simulation.txt` that you will write -- one value per line.

- Use an STL `queue` to represent the single **wait queue**. The queue should store customer objects. Use an STL `priority_queue` to store the service events that get scheduled each time service begins. Use a stack, queue, list, or vector (your choice) to store customer objects after service is completed. It will be iterated over to determine the average time from arrival to end of service per served customer.
- Create the **server array** as a vector of customer objects to represent the customers being served by each server. When a customer is removed from the **wait queue**, you'll copy that customer to a chosen position in the **server array**. Include another corresponding vector of boolean values, set to true if the server at that index position is busy serving a customer, false otherwise, indicating that the server is idle. (There's more than one way to accomplish this, so use a different way if you wish).
- As soon as a customer starts being helped by a server, the service time interval is determined as a random number in the range from the minimum service time interval to the maximum service time interval. Add the randomly-determined service time interval to the current clock time to compute the future clock time when service will be completed. Use that information to create a service event object, and store it in the **priority queue** -- this schedules a future event when the selected server will finish serving this customer. The possible values for service time interval are whole numbers between the minimum service time and maximum service time, inclusive, all equally likely. If the minimum service time and maximum service time are the same, the service time is always the same. If the minimum service time is 1 and the maximum service time is 6, the possible service times are 1, 2, 3, 4, 5, and 6 -- all equally likely. HINT -- the last example is like rolling a 6-sided die (ref. Burns COMSC 110 textbook, ch.8).

Use this pseudocode:

```
// read the input values from a text file, one per line, in the order specified above.

// declare and create and assign arrays and queues to be used in the solution

// the clock time loop
for (int time = 0;; time++) // the clock time
{
  // handle all service events scheduled to complete at this clock time
  while (event priority queue is not empty AND clock time of the top service event equals the current clock time)
    remove the top service event from the priority queue
    determine the server associated with the removed service event
    set this server's current customer's service end time to the current clock time
    copy this server's current customer to the served customers queue
    set this server to idle

  // handle new arrivals -- can be turned away if wait queue is at maximum length!
  if time is less than the time at which new arrivals stop
    get the #of of arrivals from the Poisson process (a whole number >= 0) -- see the lecture topic notes!
    for each new arrival
      if the wait queue is full
        "turn away the customer" by adding one to the count of turned away customers
      else
        create a new customer object
        set its arrival time equal to the current clock time
        assign it an ID (A-Z)
        copy the new customer to the wait queue

  // for idle servers, move customer from wait queue and begin service
  for each server
    if (server is idle AND the wait queue is not empty)
       remove top customer from wait queue
       copy it to the server array
       set the copied customer's service time to the current clock time
       use random number generation to determing the service time interval for the customer
       create a new service event for the server, for the current clock time PLUS the service time interval
       add the service event to the event priority queue

  // output the summary
  output the current time
  output a heading for the visual prepresentation of the servers and the wait queue
  for each server
  {
    output the server's index number (zero-based)
    show the ID of the customer being served by that server (blank if idle)
    for server 0 only, show the IDs of customers in the wait queue
  }
```

```
    // summarize performance
    if time is greater than zero
      iterate over the served customers queue to get the average time between arrival and service end
        and if any, output the average of their "service end time" minus their "arrival time"
      output the rate of customers turned away as total turned away divided by the current time

    // if the user wants to quit, break out of this loop
  }
```

For debugging, you might want to put in a code block at the END of the time loop, just before the "user wants to quit" prompt, to output (1) the number of events in the event **priority queue**, and (2) if there are any events in the event **priority queue**, the service time of the top event. The number of service events should equal the number of busy (not idle) servers. The top service time should be greater than the current time. It should be the time that the next customer's service will end, and he moves from "now serving" to the **served customers queue**.

To check your "turned away per minute" calc, assume your servers are all busy and your **wait queue** is full. Calculate how many your servers can serve per minute as #of servers / ((max - min service time) / 2), and subtract that from the customer arrival rate. If it is less than zero, then your **wait queue** should not be full very often, if ever.

NOTES

1. When you use `srand` in a program, make sure to call it *only once*. The best place to put the call to srand is as the first statement in main.
2. If you model the **server array** as an array or vector of customer objects, it will *always* contain customer objects in every element. Even if there is no customer being served, there is *still* a customer object, possibly uninitialized. Remember, you cannot have a NULL object -- but you *can* have a sentinel object. Or you can use a companion boolean array to track which objects are real customers and which are not.
3. Well after the end of the simuilation, after customers have stopped arriving, the **wait queue** should be empty, and no customers should remain being served.

Submit the CPP file to the class website for credit. Do NOT submit a TXT file.

Program I/O. Input: from simulation.txt with 6 lines as described above. Output: To the console, requiring the user to press ENTER to advance the time clock in the simulation.

Sample.

```
number of servers:     4
customer arrival rate: 2.5 per minute, for 50 minutes
maximum queue length:  8
minimum service time:  3 minutes
maximum service time:  10 minutes
expected turn away rate: 1.36 per minute, while full

Time: 0
---------------------------
line now serving wait queue
---- ----------- ----------
  0       A
  1       B
  2
  3
---------------------------
Press ENTER to continue, X-ENTER to exit...

...

Time: 49
---------------------------
line now serving wait queue
---- ----------- ----------
  0       H      KPQVYD
  1       L
  2       Z
  3       F
---------------------------
Avg time: 18.962. Turned away per minute: 1.013
```

```
Press ENTER to continue, X-ENTER to exit...

Time: 50
---------------------------
line now serving wait queue
---- ----------- ----------
  0       H      PQVYDEFG
  1       L
  2       K
  3       F
---------------------------
Avg time: 18.973. Turned away per minute: 1.012
Press ENTER to continue, X-ENTER to exit...

...

Time: 100
---------------------------
line now serving wait queue
---- ----------- ----------
  0
  1
  2
  3
---------------------------
Avg time: 18.973. Turned away per minute: 0.512
Press ENTER to continue, X-ENTER to exit...
```

How to pause a console program: *show / hide*

GRADING RUBRIC *show/hide*
